Computation and Uses of the Semidiscrete Matrix Decomposition

Tamara G. Kolda
Sandia National Laboratories
and
Dianne P. O'Leary
University of Maryland

We derive algorithms for computing a semidiscrete approximation to a matrix in the Frobenius and weighted norms. The approximation is formed as a weighted sum of outer products of vectors whose elements are ± 1 or 0, so the storage required by the approximation is quite small. We also present a related algorithm for approximation of a tensor. Applications of the algorithms are presented to data compression, filtering, and information retrieval; and software is provided in C and in Matlab.

Categories and Subject Descriptors: G.1.2 [**Numerical Analysis**]: Approximation; H.3.3 [**Information Systems**]: Information Search and Retrieval

General Terms: Matrix Decomposition

Additional Key Words and Phrases: Singular Value Decomposition, Semidiscrete Decomposition, Latent Semantic Indexing, Compression

---

Kolda's work was performed while she was a postdoctoral fellow at Oak Ridge National Laboratory and was supported through the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy, under contract DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation. O'Leary's work was supported by the National Science Foundation under Grant CCR-97-32022 and by the Departement Informatik, ETH Zürich, Switzerland.
Name: Tamara G. Kolda

Address: Computational Science and Mathematics Department, Sandia National Laboratories, Livermore, CA 94551-9214, `tgkolda@sandia.gov`.

Name: Dianne P. O'Leary

Address: Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, `oleary@cs.umd.edu`.

1. INTRODUCTION

A semidiscrete decomposition (SDD) approximates a matrix as a weighted sum of outer products formed by vectors with entries constrained to be in the set $\mathcal{S} = \{-1, 0, 1\}$. O'Leary and Peleg [1983] introduced the SDD in the context of image compression, and Kolda and O'Leary [1998, 1999] used the SDD for latent semantic indexing (LSI) in information retrieval; these applications are discussed in §5.

The primary advantage of the SDD over other types of matrix approximations such as the truncated singular value decomposition (SVD) is that, as we will demonstrate with numerical examples in §7, it typically provides a more accurate approximation for far less storage.

We describe the SDD, how to calculate it, and its properties in §2. The *weighted* and *tensor* SDDs are presented in §3 and §4, respectively.

A storage-efficient implementation for the SDD is presented in §6. Numerical results with our software are presented in §7.

2. THE SDD

An SDD of an $m \times n$ matrix A is a decomposition of the form

$$A_k = \underbrace{\begin{bmatrix} x_1 \; x_2 \; \cdots \; x_k \end{bmatrix}}_{X_k} \underbrace{\begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_k \end{bmatrix}}_{D_k} \underbrace{\begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_k^T \end{bmatrix}}_{Y_k^T} = \sum_{i=1}^{k} d_i x_i y_i^T .$$

Here each x_i is an m-vector with entries from the set $\mathcal{S} = \{-1, 0, 1\}$, each y_i is an n-vector with entries from the set $\mathcal{S}$, and each d_i is a positive scalar. We call this a k-term SDD.

Although every matrix can be expressed as an mn-term SDD

$$A = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} e_i e_j^T ,$$

where e_k is the k-th unit vector, the usefulness of the SDD is in developing approximations that have far fewer terms.

Since the storage requirement for a k-term SDD is k floating point numbers plus $k(m+n)$ entries from $\mathcal{S}$, it is inexpensive to store quite a large number of terms. For example, for a dense, single precision matrix of size $10,000 \times 10,000$, almost $80,000$ SDD terms can be stored in the space of the original data, and almost $160,000$ terms can be stored for a double precision matrix of the same size.

2.1 Computing an SDD

An SDD approximation can be formed iteratively via a greedy algorithm. Let A_k denote the k-term approximation ($A_0 \equiv 0$). Let R_k be the *residual* at the kth step; that is, $R_k = A - A_{k-1}$. Then the optimal choice of the next triplet (d_k, x_k, y_k) is the solution to the subproblem

$$\min \; F_k(d, x, y) \equiv \|R_k - dxy^T\|_F^2 \quad \text{s.t. } x \in \mathcal{S}^m, \; y \in \mathcal{S}^n, \; d > 0. \tag{1}$$

This is a mixed integer programming problem. Note that if the integer constraints were replaced by $\|x\| = 1$ and $\|y\| = 1$, the solution would be the rank-1 SVD approximation to R_k.

We can simplify the optimization problem slightly as follows.

THEOREM 1. *[O'Leary and Peleg 1983] Solving the mixed integer program (1) is equivalent to solving the integer program*

$$\max\ \tilde{F}_k(x,y) \equiv \max\ \frac{(x^T R_k y)^2}{\|x\|_2^2\|y\|_2^2} \quad \text{s.t. } x \in \mathcal{S}^m,\ y \in \mathcal{S}^n. \tag{2}$$

PROOF. We can eliminate d as follows. First rewrite $F_k(d,x,y)$ as

$$F_k(d,x,y) = \|R_k\|_F^2 - 2dx^T R_k y + d^2\|x\|_2^2\|y\|_2^2. \tag{3}$$

At the optimal solution, $\partial F_k/\partial d = 0$, so the optimal value of d is given by

$$d^* = \frac{x^T R_k y}{\|x\|_2^2\|y\|_2^2}.$$

Substituting d^* in (3) yields

$$F_k(d^*,x,y) = \|R_k\|_F^2 - \frac{(x^T R_k y)^2}{\|x\|_2^2\|y\|_2^2}. \tag{4}$$

Thus solving (1) is equivalent to solving (2). □

The integer program (2) has $3^{(m+n)}$ feasible points, so the cost of an exhaustive search for the optimal solution grows exponentially with m and n. Rather than doing this, we use an *alternating algorithm* to generate an *approximate* solution to the subproblem. First y is fixed and (2) is solved for x, then that x is fixed and (2) is solved for y. The process is iterated.

Solving (2) can be done exactly when either x or y is fixed. If y is fixed, then (2) becomes

$$\max\ \frac{(x^T s)^2}{\|x\|_2^2} \quad \text{s.t. } x \in \mathcal{S}^m, \tag{5}$$

where $s = R_k y/\|y\|_2^2$. The solution to this problem can be easily computed as follows.

THEOREM 2. *[O'Leary and Peleg 1983] If the solution to the integer program (5) has exactly J nonzeros, then the solution is*

$$x_{i_j} = \begin{cases} \operatorname{sign}(s_{i_j}) & \text{if } 1 \le j \le J \\ 0 & \text{if } J+1 \le j \le m \end{cases},$$

where the elements of s in sorted order are

$$|s_{i_1}| \ge |s_{i_2}| \ge \cdots \ge |s_{i_m}|.$$

PROOF. See O'Leary and Peleg [1983]. This result is also a special case of Theorem 8. □

Thus there are only m possible x-vectors to check to determine the optimal solution for (5).

Two types of stopping criteria can be used in the alternating algorithm for the solution of (2). Since from (4)

$$\|R_{k+1}\|_F^2 = \|R_k\|_F^2 - \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2}, \tag{6}$$

the inner iteration can be stopped when

$$\beta \equiv \frac{(x^T R_k y)^2}{\|x\|_2^2 \|y\|_2^2},$$

becomes nearly constant. Alternatively, a maximum number of inner iterations can be specified. These two stopping criteria can be used in conjunction.

As long as the inner iterations are terminated whenever a fixed point is encountered, the inner loop is guaranteed to be finite since no iteration makes the residual larger and there are only a finite number of possible vectors x and y.

Figure 1 shows the algorithm to generate an SDD approximation. The method will generate an approximation A_k for which $k = k_{\max}$ or $\|A - A_k\| < \rho_{\min}$. The work of each inner iteration is controlled by the parameters $l_{\max}$, the maximum number of allowed inner iterations, and $\alpha_{\min}$, the relative improvement threshold. The approximation A_k in Step (2e) is usually not formed explicitly; rather, the individual elements (d_k, x_k, y_k) are stored. Similarly, R_{k+1} in Step (2f) can be applied in Steps (2(b)i), (2(b)ii), (2(b)iii), and (2d) without explicitly forming it.

2.2 Convergence of the SDD

We show that the norm of the residual generated by the SDD algorithm is strictly decreasing and that under certain circumstances, the approximation generated by the SDD algorithm converges linearly to the original matrix.

LEMMA [O'LEARY AND PELEG 1983]. *The residual matrices generated by the SDD algorithm satisfy*

$$\|R_{k+1}\|_F < \|R_k\|_F \text{ for all } k \text{ such that } R_k \neq 0.$$

PROOF. At the end of the inner iterations, we are guaranteed to have found x_k and y_k such that $x_k^T R_k y_k > 0$. The result follows from (6). □

Several strategies can be used to initialize y in Step (2a) in the SDD algorithm (Figure 1):

(1) *Maximum element* (MAX) initialization initializes $y_k = e_j$, where j is the column containing the largest magnitude entry in R_k. The MAX initialization scheme leads to a linearly convergent algorithm (Theorem 3) but is computationally expensive if R_k is stored implicitly as $A - A_k$.

(2) *Cycling* (CYC) initialization sets $y_k = e_i$ where $i = (k \bmod n) + 1$. Unfortunately, the rate of convergence can be as slow as n-step linear [Kolda 1997].

(3) *Threshold* (THR) initialization also cycles through the unit vectors, but it does not accept a given vector unless it satisfies $\|R_k e_j\|_2^2 \geq \|R_k\|_F^2/n$. We are guaranteed that at least one unit vector will satisfy this inequality by definition of the F-norm. Even though R_k is stored implicitly, this threshold test is easy to perform because we only need to multiply R_k by a vector. Furthermore, if

(1) Let R_k denote the residual, and initialize $R_1 \leftarrow A$.
Let $\rho_k = \|R_k\|_F^2$ be the norm of the residual, and initialize $\rho_1 \leftarrow \|R_1\|_F^2$.
Let A_k denote the k-term approximation, and initialize $A_0 \leftarrow 0$.
Choose $k_{\max}$, the maximum number of terms in the approximation.
Choose $\rho_{\min}$, the desired accuracy of the approximation.
Choose $l_{\max}$, the maximum allowable inner iterations.
Choose $\alpha_{\min}$, the minimum relative improvement, and set $\alpha > 2\alpha_{min}$.

(2) For $k = 1, 2, \ldots, k_{\max}$, while $\rho_k > \rho_{\min}$, do

(a) Choose y so that $R_k y \neq 0$.

(b) For $l = 1, 2, \ldots, l_{\max}$, while $\alpha > \alpha_{\min}$, do

i. Set $s \leftarrow \dfrac{R_k y}{\|y\|_2^2}$.
Solve $\max \dfrac{(x^T s)^2}{\|x\|_2^2}$ s.t. $x \in \mathcal{S}^m$.

ii. Set $s \leftarrow \dfrac{R_k^T x}{\|x\|_2^2}$.
Solve $\max \dfrac{(y^T s)^2}{\|y\|_2^2}$ s.t. $y \in \mathcal{S}^n$.

iii. $\beta \leftarrow \dfrac{(x^T R_k y)^2}{\|x\|_2^2 \|y\|_2^2}$.

iv. If $l > 1$: $\alpha \leftarrow \dfrac{\beta - \bar{\beta}}{\bar{\beta}}$.

v. $\bar{\beta} \leftarrow \beta$.

End l-loop.

(c) $x_k \leftarrow x$, $y_k \leftarrow y$.

(d) $d_k \leftarrow \dfrac{x_k^T R_k y_k}{\|x_k\|_2^2 \|y_k\|_2^2}$.

(e) $A_k \leftarrow A_{k-1} + d_k x_k y_k^T$.

(f) $R_{k+1} \leftarrow R_k - d_k x_k y_k^T$.

(g) $\rho_{k+1} \leftarrow \rho_k - \beta$.

End k-loop.

Fig. 1. Computing an SDD

the first vector tried is accepted, no extra computational expense is incurred because the computed vector $s = R_k y$ is used in the inner iteration. This scheme is shown to be linearly convergent (Theorem 4).

(4) *SVD* initialization uses a discrete version of the left singular vector v of R_k, corresponding to the largest singular value, to initialize the iteration. If the integer restriction on our problem (1) is removed, then the singular vector is optimal, and we can form a discrete approximation to it by finding $y \in \mathcal{S}^n$ that is a discrete approximation to v; that is, find a y that solves

$$\min \|\hat{y} - v\|_2 \quad \text{s.t. } y \in \mathcal{S}^n, \hat{y} \equiv y/\|y\|_2. \tag{7}$$

This also yields a linearly convergent algorithm (Theorem 6).

We conclude this section with the proof of these convergence results.

THEOREM 3. *[Kolda 1997] The sequence $\{A_k\}$ generated by the SDD algorithm with MAX initialization converges to A in the Frobenius norm. Furthermore, the rate of convergence is at least linear.*

PROOF. Without loss of generality, assume that $R_k \neq 0$ for all k; otherwise, the algorithm terminates at the exact solution. Consider a fixed index k, and let (i, j) be the index of the largest magnitude element in R_k. Then the MAX initialization scheme chooses $y = e_j$. Since the first part of the inner iteration picks the optimal x, it must be as least as good as choosing $x = e_i$, so

$$\frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} \geq \frac{(e_i^T R_k e_j)^2}{\|e_i\|_2^2 \|e_j\|_2^2} \geq r_{ij}^2 \geq \frac{\|R_k\|_F^2}{mn}. \tag{8}$$

Thus

$$\|R_{k+1}\|_F^2 = \|R_k\|_F^2 - \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} \leq \left(1 - \frac{1}{mn}\right) \|R_k\|_F^2 \leq \left(1 - \frac{1}{mn}\right)^k \|R_0\|_F^2.$$

Hence $\|R_k\|_F \to 0$, and the rate of convergence is at least linear. □

THEOREM 4. *[Kolda 1997] The sequence $\{A_k\}$ generated by the SDD algorithm with THR initialization converges to A in the Frobenius norm. Furthermore, the rate of convergence is at least linear.*

PROOF. The proof is similar to that for Theorem 3 and so is omitted. □

Using arguments similar to those in Theorem 2, we can see that the discretization of the singular vector for SVD initialization can be computed easily.

THEOREM 5. *[Kolda 1997] For the integer program (7), if it is known that y has exactly J nonzeros, then the closest $y \in \mathcal{S}^n$ to v is given by*

$$y_{i_j} = \begin{cases} \mathit{sign}(v_{i_j}) & \text{if } 1 \leq j \leq J \\ 0 & \text{if } J+1 \leq j \leq n \end{cases},$$

where the elements of v have been sorted so that

$$|v_{i_1}| \geq |v_{i_2}| \geq \cdots \geq |v_{i_m}|.$$

Therefore, there are only n possible y-vectors to check to determine the optimal solution for (7).

THEOREM 6. *[Kolda 1997] The sequence $\{A_k\}$ generated by the SDD algorithm with SVD initialization converges to A in the Frobenius norm. Furthermore, the rate of convergence is at least linear.*

PROOF. Let (σ, u, v) be the first singular triplet of R_k. Denote the (i, j) entry of R_k by r_{ij}. Choose an initial y that solves (7). Without loss of generality, assume that the elements of v are ordered so that

$$|v_1| \geq |v_2| \geq \ldots \geq |v_n|.$$

Let J be the number of nonzeros in y. Then

$$\sigma = u^T R v = \sum_{j=1}^{J} v_j \sum_{i=1}^{m} r_{ij} u_i + \sum_{j=J+1}^{n} v_j \sum_{i=1}^{m} r_{ij} u_i,$$

and the largest magnitude elements of v must correspond to the largest magnitude elements of Ru (since $v = \sigma Ru$), so

$$\sum_{j=1}^{J} v_j \sum_{i=1}^{m} r_{ij} u_i \geq \frac{J}{n} \sigma.$$

Each v_i is less than or equal to one in magnitude, so substituting y in place of v yields

$$\sum_{j=1}^{J} y_j \sum_{i=1}^{m} r_{ij} u_i = \sum_{i=1}^{m} u_i \sum_{j=1}^{J} r_{ij} y_j \geq \frac{J\sigma}{n}.$$

(Note that this guarantees that $Ry \neq 0$.) Thus there exists $\hat{\imath}$ such that

$$u_{\hat{\imath}} \sum_{j=1}^{n} r_{\hat{\imath}j} y_j \geq \frac{J\sigma}{mn}.$$

Therefore, setting $x = e_{\hat{\imath}}$ gives

$$\frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} \geq \frac{J^2 \sigma^2}{J^2 m^2 n^2} \geq \frac{\sigma^2}{m^2 n^2} = \frac{\|R_k\|_F^2}{\min\{m, n\} \cdot m^2 n^2}. \tag{9}$$

The proof concludes using the same arguments as in Theorem 3. □

An implementation discussion is given in §6, and numerical comparisons of the different initialization strategies are presented in §7.

3. THE WEIGHTED SDD

Let $A \in \Re^{m \times n}$ be a given matrix, and let $W \in \Re^{m \times n}$ be a given matrix of nonnegative weights. The *weighted approximation problem* is to find a matrix $B \in \Re^{m \times n}$ that solves

$$\min \ \|A - B\|_W^2,$$

subject to some constraints on B. Here the *weighted norm* $\| \cdot \|_W$ is defined as

$$\|A\|_W^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 w_{ij}.$$

The case where B is a low rank matrix has been considered by Gabriel and Zamir [1979] and others, and they obtain a solution with some similarities to the truncated singular value decomposition, although computation is much more expensive. We show how to generate a weighted approximation of the form dxy^T. As with the regular SDD, we form the 1-term approximations iteratively and add these approximations together to build up a k-term approximation. At each step, then, we solve the problem

$$\min\ F_k(d,x,y) \equiv \|R_k - dxy^T\|_W^2 \quad \text{s.t. } x \in \mathcal{S}^m,\ y \in \mathcal{S}^n,\ d > 0. \tag{10}$$

Here $R_k \equiv A - \sum_{i=1}^{k-1} d_i x_i y_i^T$ is the residual matrix. As with the regular SDD, this is a mixed integer programming problem that can be rewritten as an integer program. First, recall the definition of the *Hadamard* or *elementwise product* of matrices; that is, $(A \circ B)_{ij} = a_{ij} b_{ij}$.

THEOREM 7. *Solving the mixed integer program (10) is equivalent to solving the integer program*

$$\max\ \tilde{F}_k(x,y) \equiv \frac{\left[x^T (R_k \circ W)\, y\right]^2}{(x \circ x)^T\, W\, (y \circ y)} \quad \text{s.t. } x \in \mathcal{S}^m,\ y \in \mathcal{S}^n. \tag{11}$$

PROOF. The proof is analogous to that of Theorem 1 except that

$$d^* = \frac{x^T (R_k \circ W)\, y}{(x \circ x)^T\, W\, (y \circ y)}.$$

□

As with the regular SDD, an alternating method will be used to generate an approximate solution to (11). Assuming that y is fixed, $\tilde{F}_k$ can be written as

$$\tilde{F}_k(x,y) = \frac{\left(x^T s\right)^2}{(x \circ x)^T v}, \tag{12}$$

where $s \equiv (R_k \circ W)\, y$ and $v \equiv W\,(y \circ y)$. To determine the maximum, $2^m - 1$ possibilities must be checked. Again, this can be reduced to just checking m possibilities, although the proof is more difficult than that for Theorem 2.

THEOREM 8. *For the integer program (12), if it is known that x has exactly J nonzeros, then the solution is given by*

$$x_{i_j} = \begin{cases} \text{sign}(s_{i_j}) & \text{if } 1 \le j \le J \\ 0 & \text{if } J+1 \le j \le m \end{cases},$$

where the pairs of (s_i, v_i) have been sorted so that

$$\frac{|s_{i_1}|}{v_{i_1}} \ge \frac{|s_{i_2}|}{v_{i_2}} \ge \cdots \ge \frac{|s_{i_m}|}{v_{i_m}}.$$

PROOF. First note that if s_i is zero, then a nonzero value of x_i cannot affect the numerator of $\tilde{F}$, and $x_i = 0$ minimizes the denominator, so $x_i = 0$ is optimal. If $v_i = 0$, then $s_i = 0$, so choose $x_i = 0$. Therefore, we need only consider indices for which s_i and v_i are nonzero, and without loss of generality, we will assume that the s_i are all positive and ordered so that $i_j = j$, $j = 1, \ldots, m$.

(1) Let R_k denote the residual, and initialize $R_1 \leftarrow A$.
Let $\rho_k = \|R_k\|_W^2$ be the norm of the residual, and initialize $\rho_1 \leftarrow \|R_1\|_W^2$.
Let A_k denote the k-term approximation, and initialize $A_0 \leftarrow 0$.
Choose $k_{\max}$, the maximum number of terms in the approximation.
Choose $\rho_{\min}$, the desired accuracy of the approximation.
Choose $l_{\max}$, the maximum allowable inner iterations.
Choose $\alpha_{\min}$, the minimum relative improvement, and set $\alpha > 2\alpha_{min}$.

(2) For $k = 1, 2, \ldots, k_{\max}$, while $\rho_k > \rho_{\min}$, do

(a) Choose y so that $(R_k \circ W)y \neq 0$..

(b) For $l = 1, 2, \ldots, l_{\max}$,while $\alpha > \alpha_{\min}$, do

i. Set $s \leftarrow (R_k \circ W)\,y$, $v \leftarrow W\,(y \circ y)$.
Solve $\max \dfrac{(x^T s)^2}{(x \circ x)^T v}$ s.t. $x \in \mathcal{S}^m$.

ii. Set $s \leftarrow (R_k \circ W)^T x$, $v \leftarrow W^T(x \circ x)$.
Solve $\max \dfrac{(y^T s)^2}{\|y\|_2^2}$ s.t. $y \in \mathcal{S}^n$.

iii. $\beta \leftarrow \dfrac{\left[x^T(R_k \circ W)\,y\right]^2}{(x \circ x)^T W\,(y \circ y)}$.

iv. If $l > 1$: $\alpha \leftarrow \dfrac{\beta - \bar{\beta}}{\bar{\beta}}$.

v. $\bar{\beta} \leftarrow \beta$.

End l-loop.

(c) $x_k \leftarrow x$, $y_k \leftarrow y$.

(d) $d_k \leftarrow \dfrac{x_k^T(R_k \circ W)\,y_k}{(x \circ x)^T W\,(y \circ y)}$.

(e) $A_k \leftarrow A_{k-1} + d_k x_k y_k^T$.

(f) $R_{k+1} \leftarrow R_k - d_k x_k y_k^T$.

(g) $\rho_{k+1} \leftarrow \rho_k - \beta$.

End k-loop.

Fig. 2. Weighted SDD Algorithm

We complete the proof by showing that if the optimal solution has nonzeroes with indices in some set I, and if $q \in I$ and $p < q$, then $p \in I$.

Assume to the contrary, and partition I into $I_1 \cup I_2$, where indices in I_1 are less than p and those in I_2 are greater than p. The case $p = 1$ is left to the reader; here we assume $p > 1$.

For ease of notation, let

$$S_1 = \sum_{i \in I_1} s_i, \quad V_1 = \sum_{i \in I_1} v_i,$$

and define S_2 and V_2 analogously.

By the ordering of the ratios s/v, we know that $s_i v_p < s_p v_i$ for all $i \in I_2$; therefore,

$$S_2 v_p < s_p V_2 \,. \tag{13}$$

Since I is optimal, we know that

$$\frac{S_1^2}{V_1} \le \frac{(S_1 + S_2)^2}{V_1 + V_2} \,;$$

therefore, by cross-multiplying and canceling terms, we obtain

$$S_1^2 V_2 \le S_2^2 V_1 + 2 S_1 S_2 V_1 \,. \tag{14}$$

Similarly,

$$\frac{(S_1 + S_2 + s_p)^2}{V_1 + V_2 + v_p} \le \frac{(S_1 + S_2)^2}{V_1 + V_2} \,,$$

so

$$\begin{aligned} s_p^2(V_1 + V_2) + 2S_1 s_p (V_1 + V_2) + 2 S_2 s_p (V_1 + V_2) & \\ \le S_1^2 v_p + S_2^2 v_p + 2 S_1 S_2 v_p & \\ \le S_1^2 v_p + S_2 s_p V_2 + 2 S_1 s_p V_2 & \quad \text{by (13)} \\ \le (S_2^2 V_1 + 2 S_1 S_2 V_1) \frac{v_p}{V_2} + S_2 s_p V_2 + 2 S_1 s_p V_2 & \quad \text{by (14)} \\ \le (S_2 s_p V_1 + 2 S_1 s_p V_1) + S_2 s_p V_2 + 2 S_1 s_p V_2 & \quad \text{by (13)} \end{aligned}$$

Canceling terms in this inequality we obtain

$$s_p^2(V_1 + V_2) + S_2 s_p (V_1 + V_2) \le 0 \,,$$

a contradiction. □

The algorithm for the weighted SDD, shown in Figure 2, is nearly the same as the algorithm for the regular SDD, shown in Figure 1.

3.1 Convergence of the Weighted SDD

As with the regular SDD, we show that the weighted norm of the residual generated by the weighted SDD algorithm is strictly decreasing and, furthermore, the weighted SDD approximation converges linearly to the original matrix.

LEMMA 2. *The residual matrices generated by the weighted SDD algorithm satisfy*

$$\|R_{k+1}\|_W < \|R_k\|_W \text{ for all } k \text{ such that } R_k \neq 0.$$

PROOF. The proof is similar to Lemma 1 and is therefore omitted. □

As with the SDD, several different strategies can be used to initialize y in Step (2a) in the weighted SDD algorithm (Figure 2). Here, we only present the differences between these schemes and those for the SDD. The same convergence results hold, and the proofs are similar to those given for the SDD.

(1) MAX: Choose e_j such that j is the index of the column containing the largest magnitude entry in $R_k \circ R_k \circ W$.
(2) CYC: No difference.
(3) THR: Accept a given unit vector only if it satisfies $\|R_k e_j\|_W^2 \geq \|R_k\|_W^2/n$.

Note that there is no SVD starting strategy since there is no simple analog to the SVD in the weighted case.

4. THE TENSOR SDD

Let A be an $m_1 \times m_2 \times \cdots \times m_n$ *tensor* over $\Re$. The *order* of A is n. The *dimension* of A is $m \equiv \prod_{j=1}^n m_j$, and m_j is the jth *subdimension*. An element of A is specified as

$$A_{i_1 i_2 \cdots i_n},$$

where $i_j \in \{1, 2, \ldots, m_j\}$ for $j = 1, \ldots, n$. A matrix is a tensor of order two.

As with matrices, we may be interested in a storage-efficient approximation of a given tensor. We extend the notion of the SDD to a *tensor* SDD. First we define some notation for tensors, consistent with [Kolda 1999].

4.1 Notation

If A and B are two tensors of the same size (that is, the order n and all subdimensions m_j are equal), then the *inner product* of A and B is defined as

$$A \cdot B \equiv \sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_n=1}^{m_n} A_{i_1 i_2 \cdots i_n} B_{i_1 i_2 \cdots i_n}.$$

We define the *norm* of A, $\|A\|$, to be

$$\|A\|^2 \equiv A \cdot A = \sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_n=1}^{m_n} A_{i_1 i_2 \cdots i_n}^2.$$

Suppose B is an $m_1 \times \cdots \times m_{j-1} \times m_{j+1} \times \cdots \times m_n$ tensor of order $n-1$. Then the i_jth ($1 \leq i_j \leq m_j$) element of the *contracted product* of A and B is defined as

$$(A \cdot B)_{i_j} \equiv \sum_{i_1=1}^{m_1} \cdots \sum_{i_{j-1}=1}^{m_{j-1}} \sum_{i_{j+1}=1}^{m_{j+1}} \cdots \sum_{i_n=1}^{m_n} A_{i_1 \cdots i_{j-1} i_j i_{j+1} \cdots i_n} B_{i_1 \cdots i_{j-1} i_{j+1} \cdots i_n}.$$

A *decomposed* tensor is a tensor that can be written as

$$x = x^{(1)} \otimes x^{(2)} \otimes \cdots \otimes x^{(n)},$$

where $x^{(j)} \in \Re^{m_j}$ for $j = 1, \ldots, n$. The vectors $x^{(j)}$ are called the *components* of x. In this case,

$$x_{i_1 i_2 \cdots i_n} = x^{(1)}_{i_1} x^{(2)}_{i_2} \cdots x^{(n)}_{i_n}.$$

Lowercase letters denote decomposed tensors.

Lemma [Kolda 1999]. *Let A be a tensor of order n and x a decomposed tensor of order p. Then*

$$A \cdot x = (A \cdot x^{(-j)}) \cdot x^{(j)},$$

where the notation $x^{(-j)}$ indicates x with the jth component removed, that is,

$$x^{(-j)} \equiv x^{(1)} \otimes \cdots \otimes x^{(j-1)} \otimes x^{(j+1)} \otimes \cdots \otimes x^{(p)}.$$

The notion of rank for tensors of order greater than two is a nontrivial matter (see, e.g., Kolda [1999]), but a single decomposed tensor is always a tensor of rank one.

4.2 Definition of the Tensor SDD

Suppose we wish to approximate an n-dimensional tensor A as follows,

$$A \approx A_k \equiv \sum_{i=1}^{k} d_i x_i,$$

where $d_i > 0$ and x_i is a decomposed tensor whose components are restricted to $x_i^{(j)} \in \mathcal{S}^{m_j}$, with $\mathcal{S} = \{-1, 0, 1\}$. This is called a k-term *tensor SDD*.

The SDD representation is efficient in terms of storage. If the tensor A is dense, the total storage required for A is

$$\gamma \prod_{j=1}^{n} m_j,$$

where γ is the amount of storage required for each element of A. For example, if the elements of A are integer values between 0 and 255, then γ is one byte (8 bits). The storage required for the approximation A_k is

$$k \left(\alpha + \beta \sum_{j=1}^{n} m_j \right),$$

where α is the storage required for each d_k and is usually chosen to be equal to γ and β is the amount of storage required to store each element of $\mathcal{S}$, that is, $\log_2 3$ bits. Since $k \ll \prod_{j=1}^{n} m_j$, the approximation generally requires significantly less storage than the original tensor.

4.3 Computing a Tensor SDD

As with the regular and weighted SDDs, a tensor SDD can be constructed via a greedy algorithm. Each iteration, a new d and x are computed that are the solution to the following subproblem:

$$\min \; F_k(d, x) \equiv \|R_k - dx\|^2 \quad \text{s.t. } d > 0, \; x^{(j)} \in \mathcal{S}^{m_j} \text{ for } j = 1, \ldots, n, \tag{15}$$

where $R_k \equiv A - \sum_{i=1}^{k-1} d_i x_i$ denotes the kth residual matrix. This is a mixed integer programming problem, but it can be simplified to an integer program as demonstrated by the following theorem, a generalization of Theorem 1.

THEOREM 9. *Solving the mixed integer program (15) is equivalent to solving the integer program*

$$\max\ \tilde{F}(x) = \frac{(R \cdot x)^2}{\|x\|^2} \quad \text{s.t. } x^{(j)} \in \mathcal{S}^{m_j} \ \textit{for}\ j = 1, \ldots, n. \tag{16}$$

PROOF. The proof follows the same progression as the proof for Theorem 1 except that

$$d^* = \frac{R \cdot x}{\|x\|^2}.$$

□

Solving (16) is an integer programming problem that has $3^{m_1+m_2+\cdots+m_n}$ possible solutions. To solve this problem approximately, an alternating algorithm will be used. The idea is the same as for the regular and weighed SDDs. Fix all the components of x except one, say $x^{(j)}$, and find the optimal $x^{(j)}$ under those conditions. Repeat this process for another value of j, continuing until improvement in the value of $\tilde{F}(x)$ stagnates.

Assume that all components of x are fixed except $x^{(j)}$. Then (16) reduces to

$$\max \frac{(s \cdot x^{(j)})^2}{\|x^{(j)}\|_2^2} \quad \text{s.t. } x^{(j)} \in \mathcal{S}^{m_j},$$

where $s \equiv (R_k \cdot x^{(-j)})/\|x^{(-j)}\|^2$. This is same as problem (5), so we know how to solve it.

The tensor SDD algorithm is given in Figure 3. In Step (2a), x should be chosen so that $R_k \cdot x \neq 0$. Unless R_k is zero itself (in which case $A_{k-1} = A$), it is always possible to pick such an x. The for-loop in Step (2(b)i) does not need to go through the components of x in order. That loop could be replaced by "For $j = \pi(1), \pi(2), \ldots, \pi(n)$ do," where π is an n-permutation. Note that in each step of (2(b)i), the value of x may change and that the objective function is guaranteed to be at least as good as it was with the previous x.

4.4 Convergence of the Tensor SDD

Like the SDD, the tensor SDD algorithm has the property that the norm of the residual decreases each outer iteration. Furthermore, we can prove convergence results similar to those for the SDD (proofs are omitted but are similar to those for the SDD) using each of the following starting strategies in Step (2a) of the tensor SDD algorithm:

(1) MAX: Initialize $x = e_{j_1}^{(1)} \otimes e_{j_2}^{(2)} \otimes \cdots \otimes e_{j_n}^{(n)}$, where $r_{j_1 j_2 \ldots j_n}$ is the largest magnitude element of R.

(2) CYC: Same idea as for the SDD, but now the cycle is $\prod_{j=2}^{n} m_j$ long.

(1) Let R_k denote the residual, and initialize $R_1 \leftarrow A$.
Let $\rho_k = \|R_k\|^2$ be the norm of the residual, and initialize $\rho_1 \leftarrow \|R_1\|^2$.
Let A_k denote the k-term approximation, and initialize $A_0 \leftarrow 0$.
Choose $k_{\max}$, the maximum number of terms in the approximation.
Choose $\rho_{\min}$, the desired accuracy of the approximation.
Choose $l_{\max}$, the maximum allowable inner iterations.
Choose $\alpha_{\min}$, the minimum relative improvement, and set $\alpha > 2\alpha_{min}$.

(2) For $k = 1, 2, \ldots, k_{\max}$, while $\rho_k > \rho_{\min}$, do
 (a) Initialize $x = x^{(1)} \otimes x^{(2)} \otimes \cdots \otimes x^{(n)}$.
 (b) For $l = 1, 2, \ldots, l_{\max}$, while $\alpha > \alpha_{\min}$, do
  i. For $j = 1, 2, \ldots, n$ do
   Set $s \leftarrow R_k \cdot x^{(-j)}$.
   Solve $\max \dfrac{(s^T x^{(j)})^2}{\|x^{(j)}\|_2^2}$ s.t. $x^{(j)} \in \mathcal{S}^{m_j}$.
   End j-loop.
  ii. $\beta \leftarrow \dfrac{(R_k \cdot x)^2}{\|x\|^2}$.
  iii. If $l > 1$: $\alpha \leftarrow \dfrac{\beta - \bar{\beta}}{\bar{\beta}}$.
  iv. $\bar{\beta} \leftarrow \beta$.
  End l-loop.
 (c) $x_k \leftarrow x$.
 (d) $d_k \leftarrow \dfrac{R_k \cdot x_k}{\|x_k\|^2}$.
 (e) $A_k \leftarrow A_{k-1} + d_k x_k$.
 (f) $R_{k+1} \leftarrow R_k - d_k x_k$.
 (g) $\rho_{k+1} \leftarrow \rho_k - \beta$.
 End k-loop.

Fig. 3. Computing a Tensor SDD

(3) THR: Choose $x^{(-1)} = e^{(2)}_{j_2} \otimes \cdots \otimes e^{(n)}_{j_n}$ (i.e., x with the first component removed) such that

$$\|(R \cdot x^{(-1)})\|_2^2 \geq \|R\|^2 / \prod_{j=2}^{n} m_j.$$

Although an appropriate choice of $e^{(2)}_{j_2} \otimes \cdots \otimes e^{(n)}_{j_n}$ is guaranteed to exist, it may be difficult to find because of the large search space of elements to search through.

5. APPLICATIONS

The SDD is useful in applications involving storage compression, data filtering, and feature extraction. As examples, we discuss in this section the use of the SDD in image compression, chromosome classification, and latent semantic indexing of documents.

5.1 Data Compression via the SDD

If a matrix consumes too much storage space, then the SDD is one way to reduce the storage burden. For example, the SDD can be used for image compression. The SDD was originally developed by O'Leary and Peleg [1983] for this application. If each pixel value (e.g., gray level) is stored as a matrix entry, then a k-term SDD of the resulting matrix can be stored as an approximation to the original image.

Other matrix approximation techniques have been used for image compression. The SVD [Golub and Van Loan 1989] provides a set of basis vectors that gives the optimal low-rank approximation in the sense of minimizing the sum squared errors (Frobenius norm). But these vectors are expensive to generate and take quite a bit of storage space ($n+m+1$ floating point elements per term, although it is possible to use lower precision). At the other extreme, predetermined basis vectors can be used (e.g., Haar basis or other wavelet bases). In this case, the basis vectors do not need to be explicitly stored, but the number of terms is generally much larger than for the SVD. Although the SDD chooses the basis vectors to fit the particular problem (like the SVD), it chooses them with restricted entries (like the wavelet bases), making the storage per term only $\log_2 3(n+m)$ bits plus one floating point number.

Experiments using the SDD for images achieved 10 to 1 compression (using the SDD with run-length encoding) without visual degradation of the image [O'Leary and Peleg 1983].

5.2 Data Filtering via the SDD

The k-term approximations produced by the SDD algorithm can be thought of as filtered approximations, finding relations between the columns (or rows) of the matrix that are hidden by local variations. Thus, if we have many observations of the same vector-valued phenomenon, then an SDD of the data can reveal the essential unchanging characteristics.

This fact has been used in chromosome classification. Given a "training set" consisting of many observations of a given type of chromosome (e.g., a human X chromosome), an SDD of this data extracts common characteristics, similar to a principal component analysis, but typically requiring less storage space. Then the idealized representation of this chromosome can be used to identify other chromosomes of the same type (*chromosome karyotyping*). For more information on this technique, see [Conroy et al. 1999].

5.3 Feature Extraction via the SDD

The low rank approximations produced by the SDD extract features that are common among the columns (or rows) of the matrix. This task is addressed by latent semantic indexing (LSI) of documents. A database of documents can be represented by a term-document matrix, in which each matrix entry represents the importance of some term in a particular document. Documents can be clustered for retrieval based on common features. Standard algorithms use the SVD to extract these feature vectors, but the storage involved is often greater than that for the original matrix. In contrast, the SDD has been used by Kolda and O'Leary [1998, 1999] to achieve similar retrieval performance at a much lower storage cost.

6. IMPLEMENTATION DETAILS

We focus on the regular SDD; the details for the weighted and tensor SDDs are similar. The primary advantage of the SDD over matrix decompositions such as the SVD is that the SDD requires very little memory. In this section, we illustrate the data structures and implementation details of the C code in our package, SDDPACK, that achieve the storage savings.

6.1 Data Structures

An entry from the discrete set $\mathcal{S}$, referred to as an *$\mathcal{S}$-value*, can be stored using only $\log_2 3$ bits. We actually use two bits of storage per $\mathcal{S}$-value because it is advantageous in computations involving the $\mathcal{S}$-values (see §6.2) and requires only 26% more memory. The first bit is the *value* bit and is on if the $\mathcal{S}$-value is nonzero and off otherwise; the second bit is the *sign* bit and is on for an $\mathcal{S}$-value of -1, off for 1, and undefined for 0 (Table 1). The undefined bits would not be stored if we were storing using only $\log_2 3$ bits per $\mathcal{S}$-value.

Table 1. Bit representation of $\mathcal{S}$-values.

$\mathcal{S}$-Value	**Value Bit**	**Sign Bit**
0	0	undef.
1	1	0
-1	1	1

Each iteration, a new (d, x, y) triplet is computed. The x and y vectors of length m and n, respectively, are referred to as $\mathcal{S}$-vectors. In SDDPACK, we store each $\mathcal{S}$-vector's value and sign arrays packed into unsigned long integer arrays.

Suppose that we are working on a p-bit architecture (i.e., the length of a single word of memory is p bits). Then the memory allocated to the value array to hold m bits is $\lceil m/p \rceil$ words. Storage for the sign array is the same. An example of an $\mathcal{S}$-vector and its representation on an 8-bit architecture is given in Figure 4. Notice that extra bits in the last word of the array and sign bits associated with zero $\mathcal{S}$-values are undefined. Extra bits are ignored (i.e., masked to an appropriate value)

$$x = \begin{bmatrix} -1 \\ 0 \\ 1 \\ 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ -1 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

The $\mathcal{S}$-vector x shown at left has 15 entries. Let `x` be a pointer to its representation on an 8-bit architecture. In order to store 15 bits, the value and sign arrays are each allocated two words of memory. The values of all elements of the data structure are given below. The binary values of the 8-bit words are shown for the value and sign arrays; the question marks (`?`) represent bits that are undefined. Note that the *low order* bits are used first, so the representation at the word level is right-to-left.

```
x->length = 15
x->value[1] = 00011101    x->value[2] = ?0010111
x->sign[1]  = ???100?1    x->sign[2]  = ???1?100
```

Fig. 4. Illustration of `svector` data structure.

in any calculations. We used an 8-bit example for simplicity; current architectures are generally 32- or 64-bit (Table 2).

Table 2. Current architectures.

32-bit	**64-bit**
Sun Sparc	SGI Octane
Intel Pentium	Dec Alpha
IBM RS6000	

6.2 Computations with Objects Using Packed Storage

Given an $\mathcal{S}$-vector in packed storage, we can look up the value in a particular entry as follows. If i is the desired entry, then the index into the packed array is i div p, and the bit we want inside that word is i mod p, and the desired bit can be masked off. We first do a mask on the appropriate word in the value array. If the result is zero, then entry i is zero, and we need do no further work. Otherwise, the entry is either +1 or -1, and we need to determine the sign. We mask off the appropriate word in the sign array. If that is zero, the entry is +1; otherwise, it is -1.

For example, Figure 5 shows how to look up entry 10 in the example in Figure 4. Here `i` is the desired entry. To compute `index`, the index into the packed array, divide by p, the number of bits per word. Since p is always a power of two, this can be accomplished by a right shift. In this example, we right shift 3 since $\log_2 8 = 3$. Given the correct index into the packed arrays, the correct bit inside the word is determined by a mod by p. Again, since p is always a power of two, we can use a shortcut by doing a logical AND with $p - 1$, in this example, 7. Then mask the appropriate word in the value array. In this example, it is nonzero, so the entry is either +1 or -1. Then mask the appropriate word in the sign array and determine that the entry is +1.

```
i = 10
index = i >> 3
mask = 1 << (i AND 7)
x->value[index] AND mask = 00000010
x->sign[index]  AND mask = 00000000
```

Fig. 5. Looking up a value in a packed array.

Note that the alignment of the value and sign arrays makes it easy to do individual lookups of values. If we did not store the 'filler' bits in the sign array for the zero entries, the sign array would be much shorter, but we would have a difficult time knowing where in the sign array to look for the appropriate bit.

In the previous example, we saw how to look up a random entry in a packed array. Often we walk through an $\mathcal{S}$-vector in sequence. In that case, computations can be performed even more quickly by copying the current value and sign words into the register to be used p times and quickly updating the mask with just a single left shift. Every p entries, we reset the mask to one and swap the next value and sign words into the register.

The inner product between two $\mathcal{S}$-vectors, something that we require, can be computed as follows. The result is the number of nonzeros in common minus twice the number of common nonzeros with opposite signs. Pseudo-code is given in Figure 6 for the inner product of two $\mathcal{S}$-vectors `a` and `b`. In practice, the logical ANDs and ORs are done on a word-by-word basis and the popcount (sum) is determined using a lookup table on a byte-by-byte basis. So, for computing the inner product of two m-long $\mathcal{S}$-vectors, the work required is $3\lceil m/p \rceil + 4\lceil m/8 \rceil$ and requires no multiplication.

```
common = a->value AND b->value
oppsign = (a-> sign XOR b->sign) AND common
ip = popcount(common) - 2 popcount(oppsign)
```

Fig. 6. Inner product of two $\mathcal{S}$-vectors.

Each iteration of the SDD calculation, the most expensive operations are the computations of $R_k y$ or $R_k^T x$ (Steps (2(b)i) and (2(b)ii) of the SDD Algorithm of Figure 1). We focus on the computation of $R_k y$ and explain the differences for the transpose at the conclusion. The residual breaks into two parts: the original matrix, A, and the $(k-1)$-term SDD approximation that we denote by XDY^T.

The computation $v = Ay$ is a sparse matrix times an $\mathcal{S}$-vector. The sparse matrix is stored in compressed sparse column (CSC) format. We loop through the matrix columnwise, which means that we walk through the y-vector in sequence. If y_j is zero, then nothing is done with column j. Otherwise, we either add ($y_j = 1$) or subtract ($y_j = -1$) the entries in column j from the appropriate entries in the solution vector v.

The computation of $w = XDY^T y$ breaks down into three parts: $Y^T y$, $D(Y^T y)$, and $X(DY^T y)$. The first part is an $\mathcal{S}$-matrix times an $\mathcal{S}$-vector, which reduces to an inner product between two $\mathcal{S}$-vectors for each entry in the solution. The result of $Y^T y$ is an integer vector. The $D(Y^T y)$ is just a simple scaling operation, and the result is a real vector. The final product is $X(DY^T y)$, and in this case we walk through each bit in the X matrix column by column and take appropriate action. Again, only additions and subtractions are required, no multiplications.

In the case of the transpose computation, the main difference is in the computation of $A^T x$. Here, we are forced to use random access into x since A is stored in CSC format. The method for computing $(XDY^T)^T x$ is nearly identical to that described previously for $XDY^T y$, except that the roles of X and Y are swapped.

So, the only multiplications required in our computations are the diagonal scalings; everything else is additions and subtractions. Further, the pieces of the SDD are small and fit well into cache.

7. NUMERICAL RESULTS

The computational experiments presented here are done in Matlab, with the Matlab code and examples provided in SDDPACK. In general, the C SDDPACK code should be used when speed and storage efficiency are concerns. No results are presented here for the weighted or tensor SDDs although MATLAB code for these decompositions are included. No C code is provided for these in SDDPACK.

We discuss previous research and present new results on the SDD and starting criteria as well as comparisons between the SDD and the SVD.

In [Kolda 1997], comparisons of the various starting criteria on small, dense matrices are presented. To summarize, the MAX, CYC, and THR techniques are nearly identical in performance. The SVD initialization typically results in fewer inner iterations per outer iteration, but the gain is offset by the expense of computing the starting vector.

In [Kolda and O'Leary 1998], the SDD and SVD are compared for latent semantic indexing for information retrieval. At equal levels of retrieval performance, the SDD model required approximately 20 times less storage and performed queries about twice as fast. On the negative side, the SVD can be computed about four times faster than the SDD for equal performance levels. The SDD computations used option PER, as described subsequently — we may be able to improve the speed and performance by using option THR instead.

We compare various initialization strategies for the SDD on several sparse matrices from MatrixMarket; the test set is described in Table 3. We test four different initialization strategies as listed below.

THR: Cycle through the unit vectors (starting where it left off at the previous iteration) until $\|R_k e_j\|_2^2 \geq \|R_k\|_F^2/n$, and set $y = e_j$. (Threshold)

CYC: Initialize $y = e_i$, where $i = ((k-1) \bmod n) + 1$. (Cycling)

ONE: Initialize y to the all ones vector. (Ones)

PER: Initialize y to a vector such that elements $1, 101, 201, \ldots$ are one and the remaining elements are zero. (Periodic ones)

We do not test the MAX strategy because these matrices are sparse, so the residual is stored implicitly. The other parameters of the SDD are set as follows: $k_{\max}$ is set to the rank of the matrix, $\alpha_{\min} = 0.01$, $l_{\max} = 100$, and $\rho_{\min} = 0$.

The performance of these four strategies on our four test matrices is shown in Table 4. The table compares the relative reduction in the residual (as a percentage), the average number of inner iterations (which includes the extra work for initialization in THR), and the density of the final factors (as a percentage). The initialization can have a dramatic affect on the residual after k terms. In the `impcol_c` and `watson2` matrices, THR and CYC are drastically better than ONE and PER. The number of inner iterations is lowest overall for CYC, with THR being a close second. In terms of density, THR and CYC are drastically better in every case, perhaps because the initial vector is sparse. It seems that the density of the factors may be somewhat related to the density of the original matrix. Overall, THR is best, with CYC a close second.

Table 3. Test matrices.

Matrix	Rows	Cols	NNZ	Rank	Density(%)
bfw62a	62	62	450	62	11.7
impcol_c	137	137	411	137	2.2
west0132	132	132	414	132	2.4
watson2	66	67	409	66	9.2

Table 4. Comparison of initialization techniques.

bfw62a

Init.	% Resid.	In. Its.	% Density
THR	28.19	**3.69**	**9.33**
CYC	25.54	3.73	9.55
ONE	**22.86**	6.81	41.13
PER	25.48	6.79	21.48

impcol_c

Init.	% Resid.	In. Its.	% Density
THR	**3.53**	**2.58**	**1.79**
CYC	7.86	3.47	6.47
ONE	36.93	5.95	24.32
PER	31.09	6.39	21.24

west0132

Init.	% Resid.	In. Its.	% Density
THR	**0.00**	5.62	1.95
CYC	0.01	**3.25**	**1.68**
ONE	0.01	5.64	11.97
PER	0.30	8.46	3.54

watson2

Init.	% Resid.	In. Its.	% Density
THR	**16.99**	3.02	**3.87**
CYC	20.51	**2.76**	4.17
ONE	78.74	5.42	18.94
PER	75.99	4.82	10.69

In Figures 7–10, the SVD, SDD-THR, and SDD-CYC are compared. The results on `bfw62a` are given in Figure 7. The upper left plot shows a comparison of the relative residual ($\|R_k\|/\|R_0\|$) versus the number of terms. The SVD is the optimal decomposition for a fixed number of terms, so the SDD curves will lie above it. However, the SDD still gives good reduction in the residual, requiring only about twice as many terms as the SVD for the same level of reduction. SDD-THR gives a better residual than SDD-CYC until the last few terms, where SDD-CYC 'catches up'. In the upper right, a plot of the residual versus the storage is shown; for the same level of reduction in the residual, the storage requirement for the SDD is about one to two orders of magnitude less than for the SVD. In the bottom plot, the singular values and SDD values are shown, where the ith SDD value is defined as $\hat{d}_i = d_i\|x_i\|_2\|y_i\|_2$. Initially, the SDD values are smaller than the singular values because they cannot capture as much information; later, they are larger because they are capturing the information missed initially.

The `impcol_c` matrix has an interesting singular value pattern (see Figure 8): there is one isolated singular value at 11, a cluster of singular values at 3, and another cluster at 2. SDD-THR mimics the SVD closely because SDD-THR also finds one isolated singular SDD value, as many SDD values at 3, and almost as many SDD values at 2. SDD-CYC, on the other hand, has trouble mimicking singular values because it does not pick out the isolated value at first. Still, both SDD variants are superior to the SVD in terms of storage vs. residual norm.

On `west0132` (see Figure 9), we see phenomena similar to that for `impcol_c`. SDD-THR finds isolated SDD values and quickly reduces the residual — almost as quickly as the SVD itself in terms of number of terms. SDD-CYC has more trouble isolating SDD values but eventually gets them as well. Here, SDD-THR is superior to the SVD in terms of storage, but SDD-CYC is not.

The last matrix, `watson2` (see Figure 10), most closely resembles `bfw62a` in the structure of its singular values, although `watson2` has three eigenvalues that are slightly isolated, and we can see that both SDD methods eventually pick out such values which results in the steeper drops in the residual curves. Again, SDD-THR does better than the SDD-CYC in all respects. SDD-THR requires about twice as many terms to get the same reduction in storage as the SVD, while using an order of magnitude less storage.

8. CONCLUSIONS

By presenting the code for computing a SDD, we hope to stimulate more uses of this storage-efficient matrix approximation method.

SDDPACK, containing Matlab and C code for the SDD, as well as Matlab code for the weighted and tensor SDDs is available at

`http://www.cs.umd.edu/users/oleary/`.

ACKNOWLEDGMENTS

We are grateful to Professor Walter Gander and Professor Martin Gutknecht for their hospitality at ETH.

REFERENCES

Conroy, J., Kolda, T. G., and O'Leary, D. P. 1999. Chromosome identification. In preparation.

Gabriel, K. R. and Zamir, S. November 1979. Lower rank approximation of matrices by least squares with any choice of weights. *Technometrics 21*, 489–498.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations* (2nd ed.). Johns Hopkins Press.

Kolda, T. G. 1997. Limited-memory matrix methods with applications. Technical Report CS-TR-3806, Computer Science Department, University of Maryland, College Park, MD.

Kolda, T. G. 1999. Orthogonal rank decompositions for tensors. In preparation.

Kolda, T. G. and O'Leary, D. P. 1998. A semidiscrete matrix decomposition for latent semantic indexing in information retrieval. *ACM Trans. Inf. Syst. 16*, 322–346.

Kolda, T. G. and O'Leary, D. P. 1999. Latent semantic indexing via a semi-discrete matrix decomposition. In *The Mathematics of Information Coding, Extraction and Distribution*, Volume 107 of IMA Volumes in Mathematics and Its Applications, pp. 73–80. Springer-Verlag.

O'Leary, D. P. and Peleg, S. 1983. Digital image compression by outer product expansion. *IEEE Transactions on Communications 31*, 441–444.

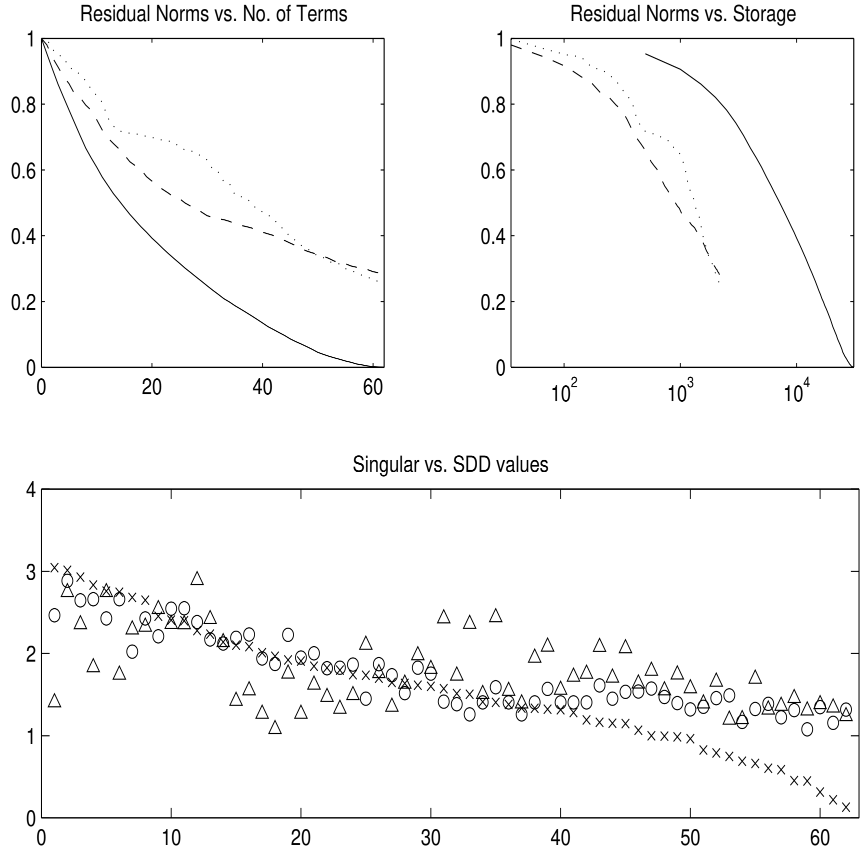

Fig. 7. Comparison of the SVD (solid line, `x` marks), SDD-THR (dashed line, `o` marks), and SDD-CYC (dotted line, triangle marks) on `bfw62a`.

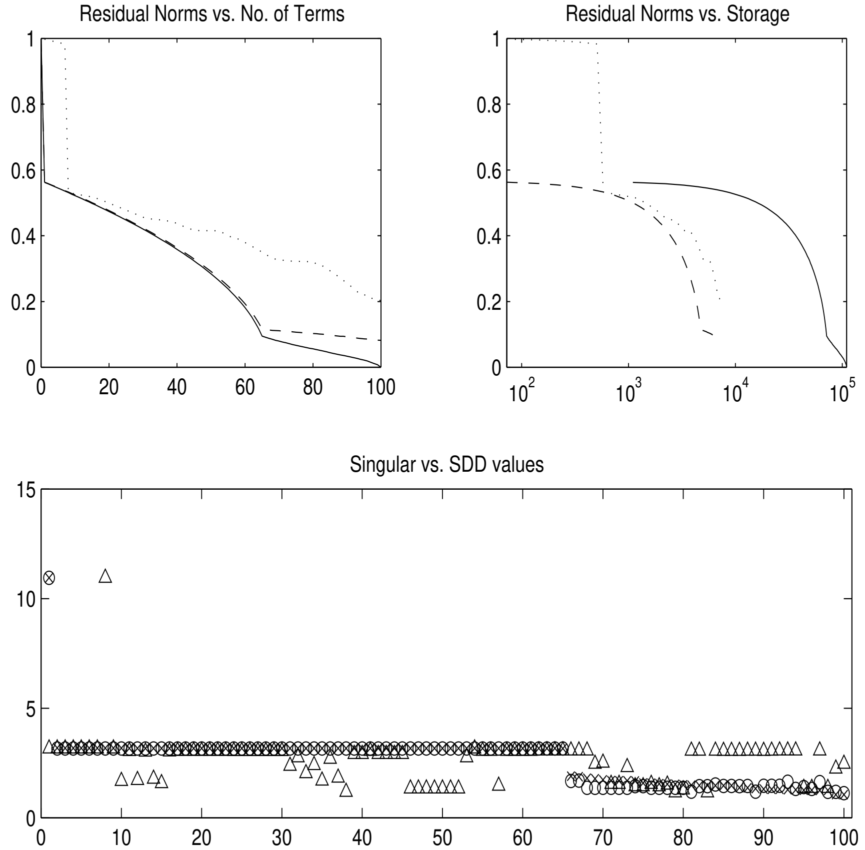

Fig. 8. Comparison of the SVD (solid line, `x` marks), SDD-THR (dashed line, `o` marks), and SDD-CYC (dotted line, triangle marks) on `impcol_c`.

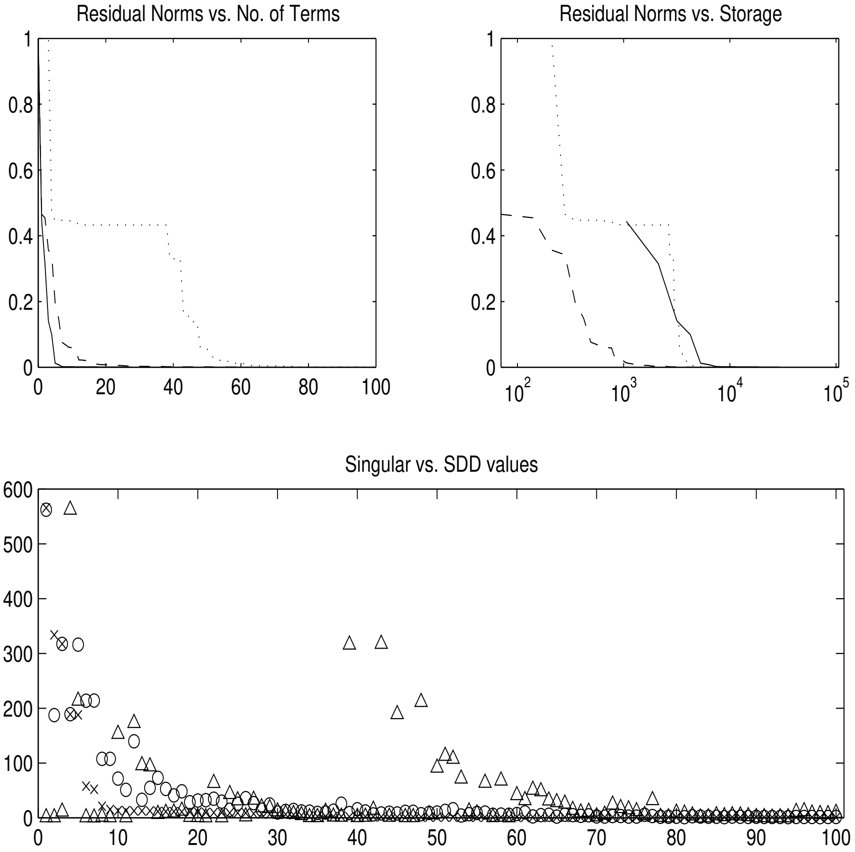

Fig. 9. Comparison of the SVD (solid line, `x` marks), SDD-THR (dashed line, `o` marks), and SDD-CYC (dotted line, triangle marks) on `west0132`.

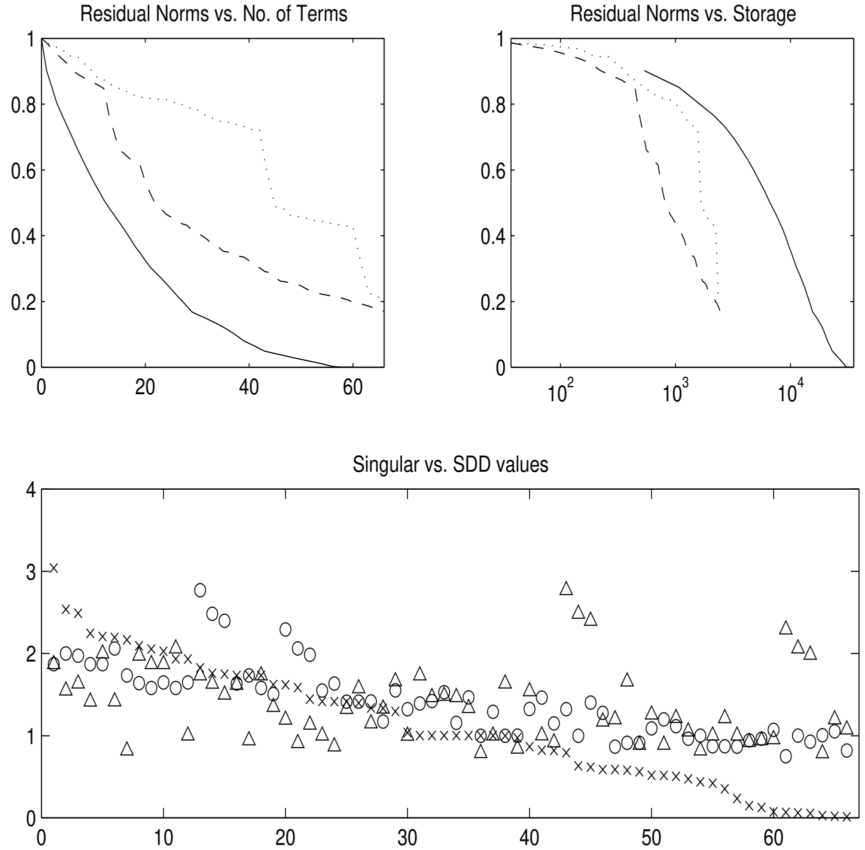

Fig. 10. Comparison of the SVD (solid line, `x` marks), SDD-THR (dashed line, `o` marks), and SDD-CYC (dotted line, triangle marks) on `watson2`.