RSVDPACK: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and GPU architectures

Sergey Voronin and Per-Gunnar Martinsson

January 19, 2015

Abstract

This document describes an implementation in C of set of randomized algorithms for computing partial Singular Value Decompositions (SVDs). The techniques largely follow the prescriptions in the article *"Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," N. Halko, P.G. Martinsson, J. Tropp, SIAM Review, 53(2), 2011, pp. 217-288* but with some modifications to improve performance on three different sets of hardware (1) single core CPU, (2) multi core CPU, and (3) massively multicore GPU. Numerical examples illustrate the performance on several different sets of test matrices.

1 Introduction

We describe a software package that computes low-rank approximations to matrices. In other words, given a $m \times n$ matrix A, loaded from a binary format on disk into RAM, we seek to compute an approximation A_{approx} of rank $k < \min(m, n)$. To be precise, the software computes a partial singular value decomposition, which is known to result in a minimal error $||A - A_{\text{approx}}||$ as measured in either the ℓ^2-operator norm, or the Frobenius norm.

The problem addressed arises frequently in scientific computing and data analysis, and also in statistics, where it is frequently referred to as "principal component analysis (PCA)."

The algorithms used are based on randomized sampling, and are highly computationally efficient, in particular in environments where reducing communication costs is a primary objective. These techniques were originally published in [3], were later extended in [2] and analyzed and surveyed in [1]. In our development, we made minor modifications to previously published versions to improve performance on specific platforms, as detailed in Section 3.

We describe three versions of the software, designed for three different hardware platforms, single core, multicore and GPU.

2 The singular value decomposition

This section introduces notation and lists some elementary facts about the SVD.

Let A be an $m \times n$ matrix with real entries. Setting $p = \min(m, n)$, every such matrix admits a so called "singular value decomposition (SVD)" of the form

$$\begin{array}{ccccc} A & = & U & \Sigma & V^T, \\ m \times n & & m \times p & p \times p & p \times n \end{array} \tag{2.1}$$

where U and V are orthonormal matrices and Σ is a diagonal matrix. The columns $(u_j)_{j=1}^p$ and $(v_j)_{j=1}^p$ of U and V are called the left and right singular vectors of A, respectively, and the diagonal entries $(\sigma_j)_{j=1}^p$ of Σ are the singular values of A. The singular values of A are ordered so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$. In other words,

$$U = \begin{bmatrix} u_1 & u_2 & \cdots & u_p \end{bmatrix}, \qquad V = \begin{bmatrix} v_1 & v_2 & \cdots & v_p \end{bmatrix}, \qquad \text{and} \qquad \Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots \\ 0 & \sigma_2 & 0 & \cdots \\ 0 & 0 & \sigma_3 & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix}.$$

It is sometimes convenient to view the factorization (2.1) as a decomposition of A as a sum of p rank-one matrices

$$A = \sum_{j=1}^{p} \sigma_j \, u_j \, v_j^T. \tag{2.2}$$

In this note, we are primarily interested in the case where the singular values σ_j decays relatively rapidly to zero, meaning that the sum (2.2) converges rapidly. In this case, it is often helpful to approximate A using an approximation $A_k \approx A$ defined by the truncated sum

$$A_k = \sum_{j=1}^{k} \sigma_j \, u_j \, v_j^T = U_k \, \Sigma_k \, V_k^T, \tag{2.3}$$

where k is a number less than p, and

$$U_k = \begin{bmatrix} u_1 & u_2 & \cdots & u_k \end{bmatrix}, \qquad V_k = \begin{bmatrix} v_1 & v_2 & \cdots & v_k \end{bmatrix}, \qquad \text{and} \qquad \Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 \\ 0 & 0 & \sigma_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & & 0 \\ 0 & 0 & 0 & \cdots & \sigma_k \end{bmatrix}.$$

It is well known that the truncated SVD A_k is the most accurate of all rank-k approximations to A, in the following sense:

Theorem 2.1 (Eckart-Young) *Let A be an $m \times n$ matrix with singular value decomposition (2.1). Then for any k such that $1 \leq k \leq \min(m, n)$, the truncated SVD A_k, as defined by (2.3) is the optimal approximation to A in the sense that*

$$||A - A_k|| = \inf\{||A - B|| : \ B \text{ has rank } k\},$$

where $||\cdot||$ is either the ℓ^2-operator norm or the Frobenius norm. Moreover,

$$||A - A_k|| = \sigma_{k+1}$$

when the error is measured in the ℓ^2 operator norm, and

$$||A - A_k|| = \left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2 \right)^{1/2}$$

when the error is measured in the Frobenius norm.

3 Randomized Algorithm

The factors U_k, Σ_k, and V_k in the partial SVD of a matrix A, cf. (2.3), can be computed by constructing the *full* SVD of A using standard software routines such as, e.g, those available in a LAPACK implementation. However, this is quite costly, with an asymptotic cost of $O(mnp)$ where $p = \min(m, n)$. The techniques presented here, in contrast, have an asymptotic cost of $O(mnk)$, which represents a substantial savings when $k \ll \min(m, n)$. To be precise, we follow the strategy based on randomized sampling outlined in [1], which roughly consists of the following steps:

- Take k samples of the range of A by multiplying A by random Gaussian vectors to form sample matrix Y of size $m \times k$ such that $\operatorname{range} Y \approx \operatorname{range} A$.
- Orthogonalize this set of samples forming the matrix Q.
- Project the original matrix into a lower dimensional one: $B = Q^T A$ where B is $k \times n$, substantially smaller than A, which is $m \times n$.
- Take the SVD of the smaller matrix $B = \tilde{U}_k \Sigma_k V_k^T$.
- Take as low rank SVD of A the product $U_k \Sigma_k V_k^T$ with $U_k = Q\tilde{U}_k$ (since $QQ^T A \approx A$).

We proceed to describe three different variants of the randomized algorithm below, which are implemented in our software package.

3.1 Version I

The first modification of the original algorithm computes U_k and V_k without having to take the SVD of the $k \times n$ matrix B. Instead, the eigendecomposition of the smaller $k \times k$ matrix BB^T is used. We use the relations:

$$B = \tilde{U}_k \Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i \tilde{u}_i v_i^T \quad ; \quad B^T = V_k \Sigma_k \tilde{U}_k^T \quad ; \quad B v_i = \sigma_i \tilde{u}_i$$

$$BB^T = \left(\sum_{i=1}^{k} \sigma_i \tilde{u}_i v_i^T \right) \left(\sum_{j=1}^{k} \sigma_j \tilde{u}_j v_j^T \right)^T = \sum_{i,j=1}^{k} \sigma_i \sigma_j \tilde{u}_i v_i^T v_j \tilde{u}_j^T = \sum_{i=1}^{k} \sigma_i^2 \tilde{u}_i \tilde{u}_i^T = \tilde{U}_k D \tilde{U}_k^T$$

To compute the right eigenvectors v_i, we can use the following relations:

$$B^T\tilde{U}_k = V_k\Sigma_k\tilde{U}_k^T\tilde{U}_k = V_k\Sigma_k \implies V_k = B^T\tilde{U}_k\Sigma^{-1}$$

assuming all the singular values in Σ_k are above zero (which is the case for k smaller than the numerical rank). The Matlab like pseudocode for this implementation proceeds as follows:

```
function [U,Sigma,V] = rsvd_version1(A,k)
    m = size(A,1);
    n = size(A,2);

    R = randn(n,k);
    Y = A*R;

    Q = orth(Y);

    B = Q'*A;
    Bt = B';

    BBt = B*Bt;

    [Uhat,D] = eig(BBt);

    Sigma = sqrt(D);
    U = Q*Uhat;

    V = Bt*Uhat*Sigma^{-1};
end
```

Note that this implementation is quite simple. The most complicated operation is the eigendecomposition of the symmetric matrix BB^T. Since this matrix is small ($k \times k$), the eigendecomposition is not expensive. The functionality to extract the eigenvalues and eigenvectors of a symmetric matrix can be found in many numerical linear algebra packages. Otherwise, one can use a Lanczos type routine to reduce the matrix to tridiagonal form and then perform repeated QR factorizations to extract the eigenvectors. However, one must be careful about orthogonality and efficiency of such implementations. The disadvantage of this version is that working with the matrix BB^T essentially squares the condition number of A, such that small singular values near machine precision may not be properly resolved. This is an issue only if A is expected to have very small singular values amongst $\sigma_1, \ldots, \sigma_k$.

3.2 Version II

Here, we use a QR factorization of B^T instead of forming BB^T. Out of this we get a $k \times k$ matrix R on which we perform the SVD. This version is based on the following calculations. Let us take the (economic form of the) QR factorization $B^T = \hat{Q}\hat{R}$. Then $\hat{R}$ is $k \times k$ and taking the SVD yields

$\hat{R} = \hat{U}\hat{\Sigma}\hat{V}^T$.

$$A \approx QQ^TA = QB = Q\hat{R}^T\hat{Q}^T = Q\hat{V}\hat{\Sigma}\hat{U}^T\hat{Q}^T$$

Thus, the low rank SVD components are $U_k = Q\hat{V}, \Sigma_k = \hat{\Sigma}, V_k = \hat{Q}\hat{U}$. The Matlab like pseudocode for this implementation proceeds as follows:

```
function [U,Sigma,V] = rsvd_version2(A,k)
    m = size(A,1);
    n = size(A,2);

    R = randn(n,k);
    Y = A*R;

    Q = orth(Y);

    Bt = A'*Q;

    [Qhat,Rhat] = qr(Bt,'0');

    [Uhat,Sigma,Vhat] = svd(Rhat);

    U = Q*Vhat;
    V = Qhat*Uhat;
end
```

3.3 Version III

This version of the algorithm is a slight modification of Version II and is effective in cases when the singular values of the matrix decrease slowly. This does not necessarily imply that the matrix is well conditioned. Rather, imagine a matrix A with singular values $\sigma_1, \ldots, \sigma_n$ where the ratio between the first and last singular value is large. However, there maybe a sequence of singular values $\sigma_{i_1}, \ldots, \sigma_{i_p}$ such that $1 \leq i_1 \leq i_p \leq n$ where the singular values fall of slowly in value. In such case, it is better to use a different method in order to approximate the range of the matrix A. In Versions I and II, the matrix Q which we use to project the matrix A into a lower dimensional space is constructed as follows:

$$Y = AR \rightarrow Q = orth(Y)$$

If we don't perform any orthogonalization, then $Q = Y$ and we sample the range of the original matrix A. Instead, based on observations in [1], we would like to work with the matrix $\left((AA^T)^qA\right)R$, where q is some integer greater than one. Note first that A and $(AA^T)^qA$ have the same eigenvectors and related eigenvalues. Plugging in the SVD $A = U\Sigma V^T$ we have:

$$AA^T = U\Sigma^2U^T \implies (AA^T)^2 = U\Sigma^2U^TU\Sigma^2U^T = U\Sigma^4U^T \implies (AA^T)^q = U\Sigma^{2q}U^T$$
$$\implies (AA^T)^qA = U\Sigma^{2q}U^TU\Sigma V^T = U\Sigma^{2q+1}V^T$$

The matrix $Z = (AA^T)^q AR$ can be built up by means of the following iterative procedure:

```
Z = A R;
for j=1:q
    Z = A A^T Z
end
```

Note that in practice, we need to orthonormalize before multiplications with A to prevent computing M^j with M a matrix having singular values greater than one. Thus, the construction of Q used to project the matrix A into a lower dimensional space can proceed as follows:

```
Y = A R
for j=1:q
    Yorth = orth(Y)
    Z = A^T Yorth
    Zorth = orth(Z)
    Y = A Zorth
end
Q = orth(Y);
```

where the orth operation is performed via a QR factorization (constructing only the factor Q, i.e. stabilized Gram-Schmidt). Note that often orthogonalization does not need be performed twice at each iteration as above. In the software, we use a parameter s which controls how often the orthogonalization is done ($s = 1$ corresponds to the above, $s = 2$ corresponds to forming Yorth but not Zorth, and so on).

4 Developed Software

In this section, we describe the developed software which has been written to implement Versions I, II, and III of the randomized SVD algorithm. We have developed three different codes: for single core, multi-core, and GPU architectures. In each case, we have used well known software libraries to implement BLAS (matrix-matrix and matrix-vector operations) and write wrappers for QR, eigendecomposition, and SVD operations. Each of the codes is written using the C programming language. Each code can load a matrix from disk stored using the following simple binary format:

```
num_rows (int)
num_columns (int)
nnz (double)
...
nnz (double)
```

where the nonzeros are listed in order of double loop over the rows and columns of the matrix. Note that even zero values are written in this format. This format is thus best for dense matrices. The matrix can be loaded using the supplied matrix_load_from_binary_file(char *fname) function. For each code, the low rank SVD can be computed by Version I, II, or III of the algorithm via the following basic call sequence (illustrated here for the single core code):

```
// load matrix
gsl_matrix *M = matrix_load_from_binary_file(mfile);
m = M->size1; n = M->size2;

// set svd rank (< min(m,n))
k = 300;

// set up SVD components
gsl_matrix *U = gsl_matrix_calloc(m,k);
gsl_matrix *S = gsl_matrix_calloc(k,k);
gsl_matrix *V = gsl_matrix_calloc(n,k);

// call random SVD (use one of three variants)
randomized_low_rank_svd1(M, k, U, S, V);
randomized_low_rank_svd2(M, k, U, S, V);
randomized_low_rank_svd3(M, k, q, s, U, S, V);

// write results to disk
matrix_write_to_binary_file(U, "data/U.bin");
matrix_write_to_binary_file(S, "data/S.bin");
matrix_write_to_binary_file(V, "data/V.bin");
```

4.1 Single Core Code

The single core code is written using functions from the GNU Scientific Library (GSL). It is a commonly used well documented library that has well optimized functions. The code uses gsl_vector and gsl_matrix objects from the library. Matrix-matrix and matrix-vector operations are done using the included BLAS functionality of GSL. For example, a call to a matrix-vector multiply looks like:

```
/* y = M*x */
void matrix_vector_mult(gsl_matrix *M, gsl_vector *x, gsl_vector *y){
    gsl_blas_dgemv (CblasNoTrans, 1.0, M, x, 0.0, y);
}
```

which is a simplified version of a regular BLAS call. Matrices and vectors are initialized via gsl_matix_set(...) and gsl_vector_set(...) commands. This code performs well and is somewhat faster than a similar Octave/Matlab implementation. However, it does not take advantage of multi-processor architectures and runs on a single core.

4.2 Multi Core Code

The multi-core code is written using the Intel Math Kernel Library (MKL) which includes multi-threaded functions for various matrix-vector operations. We define our own matrix and vector objects using the following simple constructions:

```c
typedef struct {
    int nrows, ncols;
    double * d;
} mat;

typedef struct {
    int nrows;
    double * d;
} vec;

/* initialize new matrix and set all entries to zero */
mat * matrix_new(int nrows, int ncols)
{
    mat *M = malloc(sizeof(mat));
    M->d = (double*)mkl_calloc(nrows*ncols, sizeof(double), 64);
    M->nrows = nrows;
    M->ncols = ncols;
    return M;
}

/* initialize new vector and set all entries to zero */
vec * vector_new(int nrows)
{
    vec *v = malloc(sizeof(vec));
    v->d = (double*)mkl_calloc(nrows,sizeof(double), 64);
    v->nrows = nrows;
    return v;
}
```

Access to matrix and vector elements is then provided via the functions:

```c
// column major format
void matrix_set_element(mat *M, int row_num, int col_num, double val){
    M->d[col_num*(M->nrows) + row_num] = val;
}

double matrix_get_element(mat *M, int row_num, int col_num){
    return M->d[col_num*(M->nrows) + row_num];
}


void vector_set_element(vec *v, int row_num, double val){
    v->d[row_num] = val;
}
```

```
14
15
16 double vector_get_element(vec *v, int row_num){
17     return v->d[row_num];
18 }
```

For efficiency, we avoid the use of double arrays for matrices, keeping all data in a one dimensional array. This is useful also to parallelize the code for multi-core processors. For this, we use simple OpenMP constructs. For example, the Frobenius norm of a matrix can be calculated using:

```
1  /* matrix frobenius norm */
2  double matrix_frobenius_norm(mat *M){
3      int i;
4      double val, normval = 0;
5      #pragma omp parallel shared(M,normval) private(i,val)
6      {
7      #pragma omp for reduction(+:normval)
8      for(i=0; i<((M->nrows)*(M->ncols)); i++){
9          val = M->d[i];
10         normval += val*val;
11     }
12     }
13     return sqrt(normval);
14 }
```

Here we have used OpenMP pragmas to split the work in the for loop across several different chunks, speeding up the calculation when several cores are available. The call to the low rank SVD is performed using a similar construction as in the GSL code. The code automatically distributes work amongst the number of available cores. Alternatively one can specify the number of threads to use via an environmental variable setting:

```
1 export OMP_NUM_THREADS=6
```

The number of threads should generally be set to the number of available cores for best performance. This code is significantly faster than the single core code for larger matrices.

4.3 GPU Code

The GPU code is written using the CULA library which implements BLAS and LAPACK functionality using the CUDA toolkit for NVIDIA graphics cards. CULA provides two versions of each function: a version which transfers data between host and GPU and performs calculation on the GPU, then transfers the result back to the host; and a device version, which assumes data is already on the GPU and keeps the result on the GPU after computation. The default code uses the host functions and openMP host code. That is for example, the Frobenius norm of the matrix can still be performed with openMP multi-threading on the host. On the other hand expensive functions like QR and SVD

are evaluated on the GPU. We have also implemented an experimental device code, which moves data manually between CPU and GPU and calls the device versions of the CULA functions. We have found that this strategy does not offer any noticeable performance improvements and hence make available only the host code in our software distribution.

5 Performance Comparisons

We now present some performance comparisons. We run on a high-end 6 core PC containing an Intel Xeon E5-2440 chip (6 cores, up to 2.90 Ghz) and an NVIDIA Tesla M2070-Q graphics card. We record the following runtimes for a dense 2000×4000 matrix (below, we present results for the 3 codes using version I and version II of the randomized algorithm):

k	Single Core (s)	Multi Core (s)	GPU (s)
300	6	< 1	< 1
600	18	1	1
1200	52	2	2

Runtimes for V1 on Cluster

k	Single Core (s)	Multi Core (s)	GPU (s)
300	7	< 1	< 1
600	22	1	1
1200	83	2	2

Runtimes for V2 on Cluster

For a larger $6000x12000$ dense matrix, we record the following runtimes:

k	Single Core (s)	Multi Core (s)	GPU (s)
1500	260	13	4
3000	> 300	31	11
6000	> 300	72	43

Runtimes for V1

k	Single Core (s)	Multi Core (s)	GPU (s)
1500	273	12	5
3000	> 300	34	15
6000	> 300	138	65

Runtimes for V2

The CULA GPU code offers even better performance then the MKL multi-core code for Version III of the algorithm, when $q >> 1$. Below, we present the runtimes for the same 6000×12000 matrix as before with $k = 1500$ and different values of q (and $s = 1$, for the most re-orthogonalizations between matrix multiplications).

q	Multi Core (s)	GPU (s)
5	40	12
10	65	21
20	102	38

Runtimes for V3

References

[1] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[2] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proc. Natl. Acad. Sci. USA*, 104(51):20167–20172, 2007.

[3] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the approximation of matrices. Technical Report Yale CS research report YALEU/DCS/RR-1361, Yale University, Computer Science Department, 2006.